UNIX®

SYSTEM V RELEASE 3.2

TCP/IP USER'S GUIDE AND REFERENCE

THE STANDARD FOR INTEL 386/486 PROCESSORS

intel®

Intel® TCP/IP for System V/386 Programmer's Guide and Reference

Order Number: 465729-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

For faster service in the U.S.A. or Canada, call 1-800-548-4725.

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are located directly after the reader reply card in the back of this manual.

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than the circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above	ICEVIEW	iPSB	Promware
ACE51	iCS	iPSC®	QUEST
ACE96	iDBP	iRMK®	Quick Erase
ACE186	iDIS	iRMX®	Quick-Pulse Programing
ACE196	iLBX	iSBC®	QueX
ACE960	im®	iSBX	Ripplemode
BITBUS	iMDDX	iSDM	RMX/80
COMMputer	iMMX	iSXM	RUPI
CREDIT	Inboard	Library Manager	Seamless
Data Pipeline	Insite	MAPNET	SLD
ETOX	Intel®	MCS	SugarCube
GENIUS	intel®	Megachassis	UPI
i486	Intel376	MICROMAINFRAME	VLSiCEL
i860	Intel386	MULTIBUS®	376
i	intelBOS	MULTICHANNEL	386
i	Intel Certified	MULTIMODULE	387
i®	Intelevision	ONCE	4-SITE
I2ICE	iOSP	OpenNET	486
ICE	iPAT	QTP	860
iCEL	iPDS	PROMPT	

XENIX, MS-DOS, Multiplan, and Microsoft are trademarks of Microsoft Corporation. UNIX, OPEN LOOK, and Documenter's Workbench are registered trademarks of AT&T. ETHERNET is a trademark of Xerox Corporation. Centronics is a trademark of Centronics Data Computer Corporation. Chassis Trak is a trademark of General Devices Company, Inc. VAX and VMS are trademarks of Digital Equipment Corporation, Smartmodem 1200 and Hayes are trademarks of Hayes Microcomputer Products, Inc. IBM and PC AT are registered trademarks of International Business Machines. MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation. X Window System is a trademark of Massachusetts Institute of Technology. Crystal/Writer is a registered trademark of Syntactics Corporation. ZP/ix is a registered trademark of Phoenix Technology. WEITEK is a registered trademark of MicroWay.

REV.	REVISION	DATE
-001	Original Issue.	03/90

SYSTEM V/386 TCP

PROGRAMMER'S GUIDE
AND
REFERENCE

CONTENTS

Chapter 1

SOCKET PROGRAMMER'S PRIMER

Chapter 1

PAGE

Chapter 1

SOCKET PROGRAMMER'S PRIMER

1. Introduction

This section describes the programmatic interface supported by UNIX Internetworking. The programmatic interface provides a framework for interprocess communications both within the same host and across UNIX Internetworking protocols.

2. OVERVIEW

A client process, such as a user application system, usually needs to communicate with a server process to perform its functions. One way that this interprocess communication (IPC) is provided for in UNIX is pipes. UNIX Internetworking also provides a more flexible and powerful independent subsystem especially designed to support IPC in a distributed environment. This subsystem is called the sockets interface, or "sockets." The UNIX Internetworking sockets interface comprises the programmatic interface and the basis for IPC both within a host and across an internet.

The UNIX Internetworking sockets interface is an implementation compatible with the Berkeley 4.3BSD socket mechanisms distributed for the Internet Domain. The Berkeley socket interface was designed to interface TCP/IP and other protocols with the UNIX kernel. The UNIX Internetworking version is also capable of supporting other communications protocols.

The UNIX internetworking software distribution contains the socket subsystem protocol that is necessary to support UNIX interprocess communication, networking system calls and the library subroutines.

The TCP/IP driver implements the socket abstraction and protocol. By linking their programs with the facilities in the library, **libsocket.a**, programmers can write their own distributed programs using interprocess communication. The library routines call the kernel directly. The calling programs can be written in the C language or another language.

The library, **libsocket.a**, contains system calls and library routines. The calls are linked to the actual system call primitives in the kernel. The system calls perform basic functions for an application system.

The library routines are commonly used name handling routines. The library routines are listed and described below.

3. SOCKETS

A socket is a software entity that provides the basic building block for interprocess communications. Sockets allow processes to rendezvous in a UNIX name space through which they exchange data. A socket is an endpoint of communication between processes. For Internet addresses, a fully named pair of sockets uniquely identify a connection between two communicating sides:

<<node.port> <node.port>>

where node is the four-byte network address and port is two bytes identifying the network interface. The socket on the left is the local socket and the socket on the right is the remote, or foreign, socket.

The activity status of the user processes can be seen in the Active Connections display of *netstat.* If both sides of a socket pair are operating on the local machine, each is listed separately. (See the "Network Status Monitoring" section in the TCP Administrator's Guide and Reference.

Sockets exist within communications domains. A communications domain is system of communications properties of the communicating processes and of the underlying communications facilities of the domain itself. One such property is the scheme used to name sockets. Currently UNIX supports only sockets existing in the name space of the Internet Domain. Sockets normally exchange data only with sockets in the same domain; otherwise a translation process is required.

3.1 TYPES OF SOCKETS

A socket has a type and one or more associated processes. Sockets are typed by the communications properties visible to the programmer. Usually a socket type is associated with the particular protocol which the socket supports. Processes usually communicate between sockets of the same type. Three types of sockets are available to the programmer:

- stream socket
- datagram socket
- raw socket

3.1.1 Stream Sockets

A (SOCK_STREAM) is the most commonly used type. In the AF_INET communications domain, a stream socket takes advantage of the inherent reliability of the transport level byte stream protocol, TCP. It provides bidirectional, sequenced, and unduplicated flow of data without boundaries.

3.1.2 Datagram Sockets

A datagram socket (SOCK_DGRAM) supports bidirectional flow of data in the datagram model of the network level protocol. record boundaries are preserved. The receiving process must perform resequencing, elimination of duplicates, and reliability assurance. The datagram socket can be used in applications where reliability of an individual packet is not essential, for example, in broadcasting messages for the purpose of updating a status table.

User datagram protocol (UDP) supports the datagram socket. (See the "Connectionless Sockets," section of this chapter.)

3.1.3 Raw Sockets

With a raw socket (SOCK_RAW), the programmer has access to the underlying communications protocols which support sockets, such as the IP. Raw sockets can be implemented variously depending on the interface provided by the communications protocols chosen.

Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol.

3.1.4 Other Types of Sockets

Reliable datagram and sequenced packet sockets are not available on the current version of UNIX.

4. SYSTEM CALLS

System calls are used to perform interprocess communications primarily by manipulating sockets. The linker editor, *ld*(1), searches these functions under the **-1** *socket*(2) option. The calls directly invoke UNIX system primitives in the kernel.

The UNIX networking system calls are documented in Section 2 of this manual's reference section, just as UNIX system calls are documented in section 2 of the UNIX Programmer's Reference manual. The notation:

accept(2)

tells us that **accept** has a man page in section 2 (which implies that accept is a network system call).

UNIX Networking System Calls

Call	Description
accept(2)	accept a connection on a socket
bind(2)	bind a name to a socket
connect(2)	initiate a connection on a socket
getpeername(2)	get name of connected peer
getsockname(2)	get socket name
getsockopt(2),	get options on sockets
setsockopt(2),	set options on sockets
listen(2)	listen for connections on a socket
recv(2)	receive a message from a socket
recvfrom(2)	receive a message from a socket
send(2)	send a message to a socket
sendto(2)	send a message to a socket
shutdown(2)	shut down part of a full-duplex connection
socket(2)	create an endpoint for communication
select(3)	await data or event on a socket

4.1 ERROR RETURNS

All the system calls return -1 in case of error. The error number is put in the variable *errno*. Error numbers are defined in **<sys/errno.h>**.

4.2 MODEL USAGE

In a model exchange between a calling process and a serving process, the client is the active process, the server is the passive process. The client and

the server use different types of socket calls that are appropriate to their roles. Some of the system calls that can be used are paired and arranged in logical order as follows:

Networking System Calls

Serving Process	Client Process
socket()	socket()
bind()	bind()
listen()	
accept()	connect()
read()	write()
write()	read()
close()	close()

5. CREATING A SOCKET

To create a socket, use the *socket* system call:

```
s = socket(domain, type, protocol);
```

Where

domain — In the current implementation, the domain is always AF_INET (address family_Internet domain). The manifest constants are named AF_whatever because they indicate the "address family" to use in interpreting names.

type — Types are

- SOCK_STREAM
- SOCK_DGRAM

protocol — If the protocol is unspecified (a value of 0), the system selects an appropriate protocol from those available to support the requested socket type. The system returns a small integer descriptor, or handle, to use in later system calls which operate on sockets. This is equivalent to a file descriptor. See *open*(2).

To select a particular protocol, select from those defined in **sys/in.h**. You can also use one of the library routines described below, such as *getprotobyname.*

```
Example:  s = socket(AF_INET, SOCK_STREAM,0);
```

5.1 Selecting a Protocol

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain, the available protocols are defined in **sys/in.h** or, better yet, one may use one of the library routines discussed below, such as *getprotobyname* (*getprotoent*(3)):

```
#include <sys/types.h>
#include <sys/socket.h>
#include
<netinet/in.h>
#include <netdb.h>
/* ... */
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

5.2 Socket Creation Errors

EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.
EMFILE	The per-process descriptor table is full.
ENFILE	The system file table is full.
EACCESS	Permission to create a socket of the specified type and/or protocol is denied.
ENOSR	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

5.3 BINDING SOCKET NAMES

A socket is created without a name, but, to be used, it must be given a name. The *bind* call is used to assign a name to a socket on the local side of a connection:

bind(s, name, namelen);

The argument *s* in the line above is the socket descriptor returned from the socket() call. (See "Creating a Socket," earlier in this chapter.

The argument *name* is a variable length byte string to be interpreted by the supporting protocol(s) according to the domain type. In the Internet domain, name contains an Internet address and port number, usually formatted in a structure called *sockaddr_in.*

The argument *namelen* is the length of the name.

You do not have to specify an address unless you want a certain one. To bind an Internet address, the call is:

```
#include <sys/types.h>
#include <sys/in.h>
/* ... */
struct sockaddr_in sin;
/* ... */
bind(s, &sin, sizeof (sin));
```

To determine what to place in the address *sin*, see the "Networking Library Routines" section of this chapter. If you set *sin* to 0, the system binds to the server for you and returns the address it used.

5.4 GETTING A CONNECTION

Once a process has bound a local socket, the process can rendezvous with an unrelated foreign process. Usually the rendezvous takes the form of a client server relationship.

5.4.1 The Client

The client completes the other side of the socket pair when it requests services from the server by initiating a connection by issuing a *connect* call:

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound when it issues the *connect* call, the system automatically selects a name and binds the socket. If the socket is successfully associated with the server, data transfer can begin. If not, an error is returned. Only the active process uses *connect.*

5.4.2 The Server

A completed connection is identified by a unique pair of sockets, each socket being an endpoint associated with one of the reciprocating processes. For the server to receive a client's connection, the server must issue two system calls after binding its socket. The first is to indicate a willingness to *listen* for incoming connection requests; the second is to *accept* the client's *connect.* To "listen" is to passively wait to accept a connection from a client process:

```
listen(s, 5);
```

The second parameter, 5, indicates the maximum number of outstanding connections which can be queued awaiting the acceptance of the server. This limit prevents processes from hogging system resources. Should a connection be requested while the queue is full, the server does not refuse the connection, but ignores the messages which comprise the request. This forces the client to retry the connection request and gives the server time to make room in its queue.

Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of five pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

5.4.3 Wildcard Addressing a Socket Location

A server can underspecify its location to service incoming service requests from multiple network interfaces by using the wild card symbol (*). A service such as *ftp* can installed only once on a host which is connected to multiple network interfaces *Ftp*, can *listen* on all the network interfaces:

```
<<*.21> <*.*>>
```

This tuple signifies that the local *ftp* on port 21 is *listen*ing on multiple interface addresses for whatever client processes that wish to connect.

To name a socket that listens on all network interfaces, the Internet address INADDR_ANY must be bound. If a listening port is not specified, the system assigns one. (Wildcarding is discussed further in the "IPC Programming Techniques" section of this chapter.)

5.4.4 Accepting a Connection

With the socket marked as listening, the server can now *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

The server returns a new descriptor to the client on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. *Fromlen* is a value-result parameter initialized by the server to indicate how much space is associated with *from* (the client). It is modified on return to reflect the true size of the name. Only a passive process uses *accept.*

Accept normally blocks. That is, the call to *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the *accept* call, there are alternatives which are considered in the "IPC Programming Techniques" section of this chapter.)

Servers often *bind* multiple sockets. When a server accepts a connection, it usually spins off (forks) a process which is the connected socket. The parent then goes back to listening on the same local socket.

5.4.5 Connection Errors

Of the many errors that can be returned when a connection fails, the following are the most common.

ETIMEDOUT — After failing to establish a connection during a period of time, the system decided there was no point in retrying any more. The cause for this error is usually that the remote host is down or that problems in the network resulted in transmissions being lost.

ECONNREFUSED — The host refused service for some reason. This error is usually caused by a server process not being present at the requested host.

ENETDOWN

EHOSTDOWN — Status information received by the client host from the underlying communication services indicates the net or the remote host is down.

ENETUNREACH

EHOSTUNREACH — These operational errors can occur either because the network or host is unknown (no route to the host or network is present) or because status information to that effect has been delivered to the client host by the underlying communication services.

5.5 TRANSFERRING DATA

When a connection is established, data flow can begin using a number of possible calls. If the peer entity at each end of a connection is anchored (that is, there is a connection), a user can send or receive a message, without specifying the peer, by using the *read* and *write*:

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

The calls *send* and *recv* are virtually identical to *read* and *write*, except that a flags argument is added.

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

One or more of the flags can be specified as a nonzero value as follows.

MSG_OOB — Send/receive out of band data. Out of band data is specific to stream sockets.

MSG_PEEK — Look at data without reading. When specified in a *recv* call, any data present is returned to the user but treated as though still "unread." The next *read* or *recv* call applied to the socket will return the data previously previewed.

MSG_DONTROUTE — Send data without routing packets. (Used only by the routing table management process.)

5.6 DISCARDING SOCKETS

If a socket is no longer of use, the process can discard it by applying a close to the descriptor:

```
close(s);
```

If data is associated with a socket which promises reliable delivery (a stream socket), the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. If a user process wishes to abort any pending data, it can apply a *shutdown* on the socket prior to closing it. *Shutdown* causes any data queued to be immediately discarded. The call format is:

```
shutdown(s, how);
```

where *how* is:

0	if the user no longer wishes to read data
1	if no more data will be sent
2	if no data is to be sent or received.

5.7 CONNECTIONLESS SOCKETS (SOCK_DGRAM)

UDP Datagram sockets provide only connectionless interactions. When using datagram sockets, the programmer does not have to issue a *connect* call before sending. However, in order to be informed of network level errors, it is useful to *connect*. A datagram socket provides a symmetric interface to data exchange. Datagram processes are still likely to be client and server, there is no requirement for connection establishment. Each message includes the destination address.

5.7.1 Sending from Datagram Sockets

Datagram sockets are created and name bound exactly as are stream sockets but to send data from a datagram socket, the process uses the *sendto* primitive:

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The parameters are the same as those described for *send*, above, except the *to* and *tolen* values are used to indicate the intended recipient of the message.

When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. If information at the sending node indicates that the message cannot be delivered, for instance, when a network is unreachable, the call returns -1 and the global value *errno* will contain an error number.

5.7.2 Receiving on Datagram Sockets

To receive data on an unconnected datagram socket, use *recvfrom*:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

The parameters are as described above. Note that *fromlen* is handled in the value-result manner described under "Connections," above.

5.7.3 Using Connect on a Datagram Socket

Datagram sockets can also use *connect* to associate a socket with a specific address. In this case, any data sent on the socket is automatically addressed to the connected peer, and only data received from that peer will be delivered to the user.

Only one connected address is permitted for each socket (that is, no multicasting). Connect requests return immediately; the system merely records the peer's address, as compared to a stream socket where a connect request initiates establishment of an end to end connection. Other of the less important details of datagram sockets are described in the "IPC Programming Techniques" section of this chapter.

If *connect* is used with a datagram socket, *read* and *write* and *send* and *recv* can be used to transfer data.

5.8 INPUT/OUTPUT MULTIPLEXING

An application system can multiplex I/O requests among multiple sockets by using *select*:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes three bit-masks as arguments, one for each of the following:

- the set of file descriptors for which the caller wishes to be able to read data on
- those descriptors to which data is to be written
- to indicate which exceptional conditions are pending

Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (that is, one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely. (A return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.) *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of -1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme.

6. NETWORKING LIBRARY ROUTINES

The definitive descriptions of the UNIX networking library routines are contained in their respective man pages in section 3 of this manual's reference section. All networking library routines can be distinguished by the designation 3. For example, the notation:

getservent(3)

tells us that there is a man page for the **getservent** library routine in section 3 of the reference chapter.

6.1 OVERVIEW

Most of the routines in **libsocket** are concerned with providing a uniform interface between the application system and the network database. The routines perform commonly used file name handling and manipulation. The primary uses of these routines are to locate and construct network addresses.

These routines have been designed with flexibility in mind. As more communication protocols become available, the same user interface will be maintained in accessing network-related address databases. The only difference is the values returned to the user. Since these values are normally supplied by the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; for example, "the login server on host XX". This name, and the name of the peer host, must then be translated into network addresses which are not necessarily suitable for human consumption.

Finally, the address must then used in locating a physical location and route to the service. The specifics of these three mappings is likely to vary between network architectures. The mapping process can be further complicated by additional layers required to interface disparate systems and for security considerations as described in the following example.

For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its

current location.

The UNIX networking library routines are C programming language function calls, which you can call from your program but which do not go into the kernel to be executed. They are relinked with the library at link-time. The interface can support a variety of protocols. The file **<netdb.h>** must be included when using any of these routines.

Standard routines are provided for mapping:

- host names to network addresses
- network names to network numbers
- protocol names to protocol numbers,
- service names to port numbers and the appropriate protocol to be used with the server

6.2 MAPPING HOST NAMES

A host name to address mapping is represented by the hostent data structure:

```
struct  hostent {
        char      *h_name;
        char     **h_aliases;
        int        h_addrtype;
        int        h_length;
        char     **h_addr_list;
# define           h_addr h_addr_list[0]
        }
```

where

*h_name	is the official name of the host.
**h_aliases	is the alias list.
h_addrtype	is the host address type.
h_length	is the length of address.
h_addr_list	is the address list.
h_addr	is the primary address.

The routine *gethostbyname*(3) takes a host name and returns a *hostent* structure. *Gethostbyaddr*(3) maps host addresses into a hostent structure. If a host has more than one address having the same name, *gethostbyname* returns the first entry in */etc/hosts.* If this is not adequate, the lower level routine *gethostent* can be used. An example of a routine to obtain a *hostent* structure for a host on a particular network follows:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/in.h>
#include <netdb.h>

struct hostent *
gethostbynameandnet(name,net)
char *name;
int net;
{
        register struct hostent *hp;
        register char **cp;
        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp
                            && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                        goto found;
                        continue;
                }
found:
                if ((inet_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*Inet_netof*(3) is a standard routine which returns the network portion of an Internet address. (See *inet*(3)).

6.3 MAPPING NETWORK NAMES

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

The network number is limited to 32 bits.

```
struct netent {
        char      *n_name;
        char     **n_aliases;
        int        n_addrtype;
        unsigned long        n_net;
        }
```

where

*n_name	is the official name of net.
**n_aliases	is the alias list.
n_addrtype	is the net address type.
n_net	is the network number.

The routines *getnetbyname*(3), *getnetbynumber*(3) and *getnetent*(3) are the network counterparts to the host routines described above.

6.4 MAPPING PROTOCOL NAMES

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getnetent*(3) and *getprotoent*(3):

```
struct  protoent {
        char      *p_name;
        char     **p_aliases;
        int        p_proto;
        }
```

where

*p_name	s the official protocol name.
**p_aliases	is the alias list
p_proto	is the protocol number.

6.5 MAPPING SERVICE NAMES

Information regarding services is a bit more complex. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the servent structure:

```
struct  servent {
        char       *s_name;
        char      **s_aliases;
        int         s_port;
        char       *s_proto;
        }
```

where

*s_name	is the official service name
**s_aliases	is the alias list.
s_port	is the port number.
*s_proto	is the protocol to use

The routine *getservbyname*(3) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call:

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that *telnet* server which uses the TCP protocol.

The routines *getservbyport*(3) and *getservent*(3) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

6.6 HANDLING NETWORK DEPENDENCIES

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets, there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown below:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
/* ... */
main(argc, argv)
char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        /* ... */
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service0);
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host0, argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        /* ... */
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        /* ... */
}
```

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme, within the limitations of the current organization, we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code.

6.7 Manipulating Byte Strings and Handling Byte Swapping

Aside from the address-related data base routines, there are several other routines available in the run-time library which are intended mostly to simplify manipulation of names and addresses. The following table summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1,s2,n)	compare byte-strings; 0 if same, not 0 otherwise
memcmp(s1,s2,n)	compare byte-strings
bcopy(s1,s2,n)	copy n bytes from s1 to s2
memcpy(s2,s1,c,n)	copy n bytes from s1 to s2
bzero(base,n)	zero-fill n bytes starting at base
memset(base,'\0',n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a 386 AT, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when interpreting network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d0, ntohs(sp->s_port));
```

For some architectures, such as the 3B2, these routines are defined as null macros. Nevertheless, their use is always recommended to help preserve program portability.

7. USING THE CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server. In the following paragraphs we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

8. OVERVIEW

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation the protocol may be symmetric or asymmetric.

In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the *telnet* protocol used in the internet for remote terminal emulation. An example of an asymmetric protocol is the internet file transfer protocol, *ftp*. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

9. SERVER PROCESS

A server process normally listens at a well known address for service requests. ("Well known" means that port assignments for services are usually stable and can be seen in **/etc/services**. See the "Typical TCP/IP Process" section of this chapter.) Alternative schemes which use a service server can be used to eliminate a number of server processes clogging the system while remaining dormant most of the time. For example, the Berkeley implementation of the Xerox Courier protocol uses the latter scheme. (This is an example only; UNIX does not currently support Courier.)

When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a database and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication.

However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous.

In STREAMS TCP, most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time, it advertises it services by listening at a well known location. For example, the remote login server's main loop is of the form shown below:

```
/* include header files */
main(argc, argv)
int argc;
char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        /* disassociate server from controlling terminal */

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown
                service0);
                exit(1);
        }
        sin.sin_port = sp->s_port;
        /* ... */
        f = socket(AF_INET, SOCK_STREAM, 0);
        /* ... */
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
        /* ... */
        }
        /* ... */
        listen(f, 5);
        for (;;) {
                int g, len = sizeof(struct sockaddr_in);
                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof(struct sockaddr_in);
        g = accept(f, &from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal. Therefore, the

return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

9.1 CLIENT PROCESS

The client side of the remote login service was shown above. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let's consider more closely the steps taken by the client remote login process, as in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service0);
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
p = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host0, argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the internet address of the foreign host and the port number at which the login process resides:

```
memset((char *)&sin, 0, sizeof(sin));
memcpy (hp->h_length, (char *)sin.sin_addr, hp->h_addr) ;
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated:

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
/* ... */
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

9.2 CONNECTIONLESS SERVER PROCESS

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the *rwho* service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to broadcast information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the *rwho* server may find out the current status of a machine with the *ruptime* (1) program. The output generated is illustrated below:

```
laibbb      up 21:56
laibel      up 21:54
laiden      up 18:31
laidy       up 19:29
```

Status information for each host is periodically broadcast by *rwho* server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The *rwho* server, in a simplified form, is pictured below. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the *rwho* port are interrogated to insure they have been sent by another *rwho* server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicates such on the status reports. This algorithm is prone to error since a server may be down while a host is actually up, but it works well in a LAN.

```
main()
{
        /* ... */
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        /* ... */
        s = socket(AF_INET, SOCK_DGRAM, 0);
        /* ... */
        bind(s, &sin, sizeof (sin));
        /* ... */
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);
                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port0,
                            ntohs(from.sin_port));
                        continue;
                }
                /* ... */
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x0,
                            ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O.WRONLY|O.CREATE|O.TRUNCATE, 0666);
                /* ... */
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other *rwho* servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but not overly creative. Deciding where to transmit the resultant packet does, however, indicate some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast, another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table

management process in propagating routing status information.

The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information. (One must, however, be concerned about "loops." That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.)

The second problem with the current scheme is that the *rwho* process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a problem. UNIX attempts to isolate host-specific information from applications by providing system calls which return the necessary information. (An example of such a system call is the *gethostname*(3) call which returns the host's "official" name.)

10. IPC PROGRAMMING TECHNIQUES

A number of facilities have yet to be discussed. For most users of UNIX, the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

10.1 OUT OF BAND DATA

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with a signal which, on UNIX systems, is generally defined as SIGUSR1. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the MSG_OOB flag is supplied to a send or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown below:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGUSR1, oob);

        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);

        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, MSG_OOB);
        /* ... */
}
```

10.2 SIGNAL AND PROCESS GROUPS

Because of the existence of the SIGUSR1 signal, (used where a BSD system would use the SIGURG signal), each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

10.3 PSEUDO TERMINALS

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a pseudo terminal. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side.

The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a

slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *kill(2)* on the appropriate process group.

10.4 INTERNET ADDRESS BINDING

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association.* An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The *bind* system call allows a process to specify half of an association,

```
<local address, local port>
```

while the *connect* and *accept* primitives are used to complete a socket's association. Since the association is created in two steps, the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in **sys/in.h**), the system interprets the address as "any valid address." For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <sys/in.h>
/* ... */
struct sockaddr_in sin;
/* ... */
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on a networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (that is, the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
/* ... */
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
/* ... */
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use0);
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because of associations being created in a two step process. For example, the Internet file transfer protocol, *ftp*, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's

socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected. (For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.)

10.5 BROADCASTING AND DATAGRAM SOCKETS

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htons(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = htonl(INADDR_ANY);
dst.sin_port = htons(DESTPORT);
```

Next, the socket must be set to allow the broadcasting:

```
on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the sender's address and port. (Datagram sockets are anchored before a message is allowed to go out.)

10.6 TYPICAL TCP/IP PROCESS

A calling program issues a socket call to AF_INET (Address Family Internet) to get access to the net.

TCP returns a file descriptor. If the calling program wishes a particular circuit, it looks in **/etc/services**, using a *getservent* library call, for the port assignment for the service it requires and issues a bind call to that address number. For example, the service *ftp* is commonly assigned port 21.

The program gives the port of origin and the the full destination address (family, port, host address).

(Well known services have permanently assigned ports.) TCP ports are logical ports associated with sockets, not physical communications ports. (See **/etc/services** for standard port assignments.) TCP allocates the port to the calling program. If the program does not ask to bind a port, TCP assigns one when it receives the connect call.

The calling program then issues a connect call by identifying its own full address and the full destination address. IP addresses should be retrieved through the *gethostent* call.

When the connection completes, the connection is established, the calling program writes and/or reads.

If the write exceeds the maximum packet size, TCP breaks it up into separate packets.

The receiving server does much the same thing at the other end, a socket call and a bind call. The receiving server then makes a *listen* call and an *accept*, which says, "Wait for a call, and if one comes in, give it to me." In the example of the *ftp* server, when *ftp* accepts a call, it spins off a child process which takes over the *ftp* operations. The original *ftp* demon goes back to waiting for another call. The child process takes over the new connection.

The TCP protocol on each machine keeps track of multiple concurrent sessions between machines by assigning numbered ports to the sessions. For example, session A is between machine 1.1, port 4 and machine 1.2, port 2; while session B is between machine 1.1, port 5 and machine 1.2, port 3, and so on.

11. ADDING AND CHECKING FOR SERVICES

When you need to add a new service, you must add its name to the **/etc/services** file. In your program you can issue a call to look into **/etc/services** to make sure it is on the system, before using it in your program. Refer to *getservent*(3) and *services(4)* for more information.

12. ERROR HANDLING IN PROGRAMMING

UNIX has many generic error codes. Generally, each system service maps its error condition to these codes. Some errors have been added for System V/386 Streams TCP.

You should check for error returns after every system call. Most calls have one or more error returns. These errors are described in the call description in the Reference section of this manual. An error condition is indicated by an otherwise impossible returned value. This is almost always **-1**. An error number is made available in the external variable **errno**. **Errno** is set only when an error incurred and is not cleared on successful calls, therefore test it only after an error has been indicated. Use *perror*(3) to print error messages. (For a complete list of the 88 or more error returns found in **errno.h**, see *intro*(2) in the Reference section of this manual. Note that the error numbers are subject to change.)

12.1 SOCKET ERROR INDICATIONS

If information at the sending node indicates that the message cannot be delivered, for instance, when a network is unreachable, the call returns -1 and the global value *errno* will contain an error number.

12.2 LIST OF ERROR CODES

The following is a list of internetworking error codes. The protocol or operation during which the error is likely to occur is given at the beginning of the explanation.

12.2.1 I/O Errors

EALREADY	**Operation already in progress**
EINPROGRESS	**Operation now in progress**
EWOULDBLOCK	**Operation would block**

12.2.2 Argument Errors

EADDRINUSE	**Address already in use** TCP and UDP: An attempt was made to create

a socket with a port which has already been allocated.

EADDRNOTAVAIL — **Can't assign requested address**
TCP and UDP: An attempt was made to create a socket with a network address for which no network interface exists.

EAFNOSUPPORT — **Address family not supported by protocol family**

EDESTADDRREQ — **Destination address required**

EMSGSIZE — **Message too long**

ENOTSOCK — **Socket operation on non-socket**

EOPNOTSUPP — **Operation not supported on socket**

EPFNOSUPPORT — **Protocol family not supported**

EPROTONOSUPPORT — **Protocol not supported**
Creating a Socket. Unknown protocol or protocol not supported.

EPROTOTYPE — **Protocol wrong type for socket**
Creating a Socket. Socket type request has no supporting protocol.

ESOCKTNOSUPPORT — **Socket type not supported**

12.2.3 Operational Errors

ECONNABORTED — **Software caused connection abort**

ECONNREFUSED — **Connection refused**
Socket Connection. The host refused service for some reason. This error is usually caused by a server process not being present at the requested name.

ECONNRESET — **Connection reset by peer**
TCP: The remote peer forced the session to be closed.

EISCONN — **Socket is already connected**
IP and UDP: An attempt was made to establish a connection on a socket that already has one or an attempt was made to send a datagram with

the destination address specified and the socket is already connected. TCP. An attempt was made to establish a connection on a socket that already has one.

ENETDOWN	**Network is down** Socket Connection. Status information received by the client host from the underlying communication services indicates the net or the remote host is down.
ENETRESET	**Network dropped connection on reset**
ENETUNREACH	**Network is unreachable**
ENOBUFS	**No buffer space available** TCP, IP, and UDP: Any Socket Operation. The system lacks sufficient memory for an internal data structure.
ENOTCONN	**Socket is not connected** UDP: An attempt was made to send a datagram, but no destination address is specified, and the socket has not been connected.
ESHUTDOWN	**Can't send after socket shutdown**
ETIMEDOUT	**Connection timed out** Socket Connection. After failing to establish a connection during a period of time (excessive retransmissions), the system decided there was no point in retrying any more. The cause for this error is usually that the remote host is down or that problems in the network resulted in transmissions being lost.

12.2.4 Miscellaneous Errors

EHOSTDOWN	**Host is down** Socket Connection. Status information received by the client host from the underlying communication services indicates the net or the remote host is down.
EHOSTUNREACH	**No route to host** Socket Connection. These operational errors can

	occur either because the network or host is unknown (no route to the host or network is present) or because status information to that effect has been delivered to the client host by the underlying communication services.
ENOPROTOOPT	**Protocol not available**

Chapter 2

SENDMAIL

AN INTERNETWORK ROUTER

Chapter 2

PAGE

LIST OF FIGURES

Chapter 2

SENDMAIL
AN INTERNETWORK MAIL ROUTER†

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible

---

† This chapter is based on the document of the same name, written by Eric Allman of Britton-Lee, Inc.

configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the address space.

Sendmail is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

1. DESIGN GOALS

Design goals for *sendmail* include:

1. Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

2. Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formatted addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

3. Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

4. Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

5. Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

6. *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

7. Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

8. Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1.

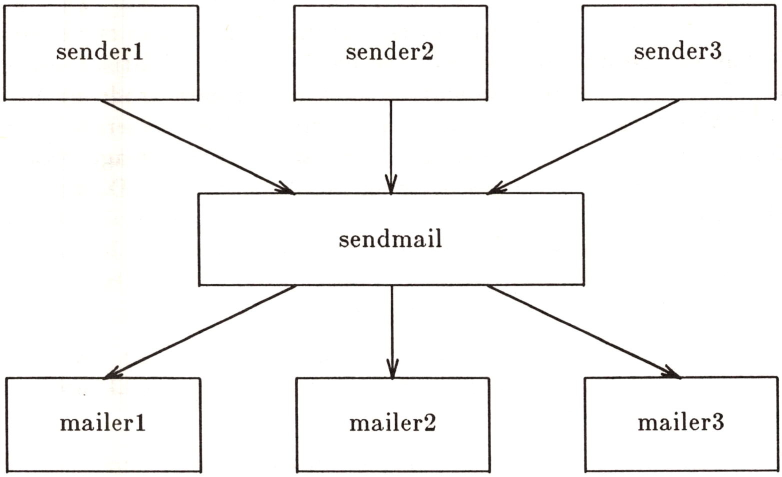

Figure 1. Sendmail System Structure

The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

2. OVERVIEW

2.1 System Organization

Sendmail neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission.[1] This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

2.2 Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

2.2.1 Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

2.2.2 SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special

1. except when mailing to a file, when *sendmail* does the delivery directly.

format.

2.2.3 SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2bsd IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a sendmail process on another machine.

2.3 Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

2.3.1 Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

Sendmail appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages delivered to the same recipient, as might occur if a person is in two groups.

2.3.2 Message collection

Sendmail then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header

is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

2.3.3 Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

2.3.4 Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

2.3.5 Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory.[2]

---

2. Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message – the "return to sender" function is always handled in one of these two ways.

2.4 Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

2.5 Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

3. USAGE AND IMPLEMENTATION

3.1 Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

1. Anything in parentheses is thrown away (as a comment).
2. Anything in angle brackets ("<>") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

   user name <machine-address>

   will send to the electronic "machine-address" rather than the human "user name."
3. Double quotes (") quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently – for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing.[3]

3.2 Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark

3. Disclaimer: Some special processing is done after rewriting local names; see below.

("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

3.3 Aliasing, Forwarding, Inclusion

Sendmail reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

3.3.1 Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

3.3.2 Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"¦/usr/local/newmail myname"

will use a different incoming mailer.

3.3.3 Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

3.4 Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

3.5 Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

3.6 Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

3.7 Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recomplied except

1. To change operating systems (V6, V7/32V, 4BSD).
2. To remove or insert the DBM (UNIX database) library.
3. To change ARPANET reply codes.
4. To add header fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

3.7.1 Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

3.7.2 Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

3.7.3 Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

3.7.4 Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

    ucsfcgl!tef

might be mapped into:

    tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

3.7.5 Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

4. COMPARISON WITH OTHER MAILERS

4.1 Delivermail

Sendmail is an outgrowth of *delivermail*. The primary differences are:

1. Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

2. Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

3. Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

4. *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

5. A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

6. *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

4.2 MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail*. For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code.[4]

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

4.3 Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples.[6] Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

---

4. Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

5. The MMDF equivalent of a *sendmail* "mailer."

6. This is similar to the NBS standard.

5. EVALUATIONS AND FUTURE PLANS

Sendmail is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

Sendmail has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

ACKNOWLEDGMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

REFERENCES

[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System.* R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility – MMDF.* 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19,* 7. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook.* NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard.* Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network.* RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol.* NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol.* RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure.* In *Proceedings of the Sixth Data Communications Symposium,* IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents.* RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell

Laboratories, modified by the University of California, Berkeley, California. March, 1983.

Chapter 3

THE BERKELEY UNIX TIME SYNCHRONIZATION PROTOCOL

Chapter 3

PAGE

Chapter 3

THE BERKELEY UNIX TIME SYNCHRONIZATION PROTOCOL†

1. Introduction

The Time Synchronization Protocol (TSP) has been designed for specific use by the program *timed,* a local area network clock synchronizer for the UNIX operating system with enhanced networking capabilities provided by System V/386 Streams TCP. Timed is built on the DARPA UDP protocol [4] and is based on a master slave scheme.

TSP serves a dual purpose. First, it supports messages for the synchronization of the clocks of the various hosts in a local area network. Second, it supports messages for the election that occurs among slave time daemons when, for any reason, the master disappears. The synchronization mechanism and the election procedure employed by the program timed are described in other documents [1,2,3].

Briefly, the synchronization software, which works in a local area network, consists of a collection of *time daemons* (one per machine) and is based on a master-slave structure. The present implementation keeps processor clocks synchronized within 20 milliseconds if supported by the hardware. Otherwise, 1 second is the best that can be done. A *master time daemon* measures the time difference between the clock of the machine on which it is running and

† This chapter is based on the document of the same name, written by Riccardo Gusella, Stefano Zatti, and James M. Bloom of the University of California at Berkeley.

those of all other machines. The current implementation uses ICMP *Time Stamp Requests* [5] to measure the clock difference between machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks.[1] It then sends to each *slave time daemon* the correction that should be performed on the clock of its machine. This process is repeated periodically. Since the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with synchronization. When a machine comes up and joins the network, it starts a slave time daemon, which will ask the master for the correct time and will reset the machine's clock before any user activity can begin. The time daemons therefore maintain a single network time in spite of the drift of clocks away from each other.

Additionally, a time daemon on gateway machines may run as a *submaster*. A submaster time daemon functions as a slave on one network that already has a master and as master on other networks. In addition, a submaster is responsible for propagating broadcast packets from one network to the other.

To ensure that service provided is continuous and reliable, it is necessary to implement an election algorithm that will elect a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned. Under our algorithm, slaves are able to realize when the master has stopped functioning and to elect a new master from among themselves. It is important to note that since the failure of the master results only in a gradual divergence of clock values, the election need not occur immediately.

All the communication occurring among time daemons uses the TSP protocol. While some messages need not be sent in a reliable way, most communication in TSP requires reliability not provided by the underlying protocol. Reliability is achieved by the use of acknowledgements, sequence numbers, and retransmission when message losses occur. When a message that requires acknowledgment is not acknowledged after multiple attempts, the time daemon that has sent the message will assume that the addressee is down. This document will not describe the details of how reliability is implemented, but will only point out when a message type requires a reliable transport mechanism.

---

1. A clock is considered to be faulty when its value is more than a small specified interval apart from the majority of the clocks of the machines on the same network. See [1,2] for more details.

The message format in TSP is the same for all message types; however, in some instances, one or more fields are not used. The next section describes the message format. The following sections describe in detail the different message types, their use and the contents of each field. NOTE: The message format is likely to change in future versions of timed.

2. Message Format

All fields are based upon 8-bit bytes. Fields should be sent in network byte order if they are more than one byte long. The structure of a TSP message is the following:

1) A one byte message type.

2) A one byte version number, specifying the protocol version which the message uses.

3) A two byte sequence number to be used for recognizing duplicate messages that occur when messages are retransmitted.

4) Eight bytes of packet specific data. This field contains two 4 byte time values, a one byte hop count, or may be unused depending on the type of the packet.

5) A zero-terminated string of up to 256 ASCII characters with the name of the machine sending the message.

The following charts describe the message types, show their fields, and explain their usages. For the purpose of the following discussion, a time daemon can be considered to be in one of three states: slave, master, or candidate for election to master. Also, the term *broadcast* refers to the sending of a message to all active time daemons.

2.1 Adjtime Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Adjustment			
Microseconds of Adjustment			
Machine Name			
. . .			

Type: TSP_ADJTIME (1)

The master sends this message to a slave to communicate the difference between the clock of the slave and the network time the master has just computed. The slave will accordingly adjust the time of its machine. This message requires an acknowledgment.

2.2 Acknowledgment Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_ACK (2)

Both the master and the slaves use this message for acknowledgment only. It is used in several different contexts, for example in reply to an Adjtime message.

2.3 Master Request Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_MASTERREQ (3)

A newly-started time daemon broadcasts this message to locate a master. No other action is implied by this packet. It requires a Master Acknowledgment.

2.4 Master Acknowledgement

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_MASTERACK (4)

The master sends this message to acknowledge the Master Request message and the Conflict Resolution Message.

2.5 Set Network Time Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
. . .			

Type: TSP_SETTIME (5)

The master sends this message to slave time daemons to set their time. This packet is sent to newly started time daemons and when the network date is changed. It contains the master's time as an approximation of the network time. It requires an acknowledgment. The next synchronization round will eliminate the small time difference caused by the random delay in the communication channel.

2.6 Master Active Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_MASTERUP (6)

The master broadcasts this message to solicit the names of the active slaves. Slaves will reply with a Slave Active message.

2.7 Slave Active Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_SLAVEUP (7)

A slave sends this message to the master in answer to a Master Active message. This message is also sent when a new slave starts up to inform the master that it wants to be synchronized.

2.8 Master Candidature Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_ELECTION (8)

A slave eligible to become a master broadcasts this message when its election timer expires. The message declares that the slave wishes to become the new master.

2.9 Candidature Acceptance Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_ACCEPT (9)

A slave sends this message to accept the candidature of the time daemon that has broadcast an Election message. The candidate will add the slave's name to the list of machines that it will control should it become the master.

2.10 Candidature Rejection Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_REFUSE (10)

After a slave accepts the candidature of a time daemon, it will reply to any election messages from other slaves with this message. This rejects any candidature other than the first received.

2.11 Multiple Master Notification Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_CONFLICT (11)

When two or more masters reply to a Master Request message, the slave uses this message to inform one of them that more than one master exists.

2.12 Conflict Resolution Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_RESOLVE (12)

A master which has been informed of the existence of other masters broadcasts this message to determine who the other masters are.

2.13 Quit Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_QUIT (13)

This message is sent by the master in three different contexts: 1) to a candidate that broadcasts an Master Candidature message, 2) to another master when notified of its existence, 3) to another master if a loop is detected. In all cases, the recipient time daemon will become a slave. This message requires an acknowledgement.

2.14 Set Date Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
. . .			

Type: TSP_SETDATE (22)

The program *date*(1) sends this message to the local time daemon when a super-user wants to set the network date. If the local time daemon is the master, it will set the date; if it is a slave, it will communicate the desired date to the master.

2.15 Set Date Request Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Seconds of Time to Set			
Microseconds of Time to Set			
Machine Name			
. . .			

Type: TSP_SETDATEREQ (23)

A slave that has received a Set Date message will communicate the desired date to the master using this message.

2.16 Set Date Acknowledgment Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
...			

Type: TSP_DATEACK (16)

The master sends this message to a slave in acknowledgment of a Set Date Request Message. The same message is sent by the local time daemon to the program *rdate(1M)* to confirm that the network date has been set by the master.

2.17 Start Tracing Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
...			

Type: TSP_TRACEON (17)

The controlling program *timedc* sends this message to the local time daemon to start the recording in a system file of all messages received.

2.18 Stop Tracing Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_TRACEOFF (18)

Timedc sends this message to the local time daemon to stop the recording of messages received.

2.19 Master Site Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_MSITE (19)

Timedc sends this message to the local time daemon to find out where the master is running.

2.20 Remote Master Site Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_MSITEREQ (20)

A local time daemon broadcasts this message to find the location of the master. It then uses the Acknowledgement message to communicate this location to *timedc*.

2.21 Test Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
(unused)			
(unused)			
Machine Name			
. . .			

Type: TSP_TEST (21)

For testing purposes, *timedc* sends this message to a slave to cause its election timer to expire. NOTE: *timed* is not normally compiled to support this.

2.22 Loop Detection Message

Byte 1	Byte 2	Byte 3	Byte 4
Type	Version No.	Sequence No.	
Hop Count	(unused)		
(unused)			
Machine Name			
. . .			

Type: TSP_LOOP (24)

This packet is initiated by all masters occasionally to attempt to detect loops. All submasters forward this packet onto the networks over which they are master. If a master receives a packet it sent out initially, it knows that a loop exists and tries to correct the problem.

3. References

1. R. Gusella and S. Zatti, *TEMPO: A Network Time Controller for Distributed Berkeley UNIX System*, USENIX Summer Conference Proceedings, Salt Lake City, June 1984.

2. R. Gusella and S. Zatti, *Clock Synchronization in a Local Area Network*, University of California, Berkeley, Technical Report, *to appear.*

3. R. Gusella and S. Zatti, *An Election Algorithm for a Distributed Clock Synchronization Program*, University of California, Berkeley, CS Technical Report #275, Dec. 1985.

4. Postel, J., *User Datagram Protocol*, RFC 768. Network Information Center, SRI International, Menlo Park, California, August 1980.

5. Postel, J., *Internet Control Message Protocol*, RFC 792. Network Information Center, SRI International, Menlo Park, California, September 1981.

Chapter 4

GLOSSARY

TCP/IP GLOSSARY

alias	An alias is an alternate host name, which can be created as a convenience in addressing a host on a local network whose unique primary name is long and/or complicated.
ARP	Address Resolution Protocol is used by Ethernet for address mapping.
ARPA	Now called DARPA, stands for Defense Advanced Research Projects Agency. ARPANET is the network based on the work sponsored by this agency. See also **DDN**
block	A block (noun) is usually 1024 bytes.
broadcast network	A broadcast network is one that "broadcasts" all transmissions instead of from point to point. Each node then "grabs" the transmissions intended for them. For example, Ethernet broadcasts down its bus.
BSD	Berkeley Software Distribution
bus	A set of parallel signals implemented in hardware in a standard manner so that multiple devices can access it and communicate over it.
connection	A connection is a logical communication path identified by a pair of sockets.
DARPA	Department of Defense Advanced Research Project Agency, formerly called ARPA. This agency sponsored the network architecture research project upon which ARPANET is based. ARPANET is a large governmental internetwork, called the Internet, part of which is the Defense Data Network (DDN). See also **DDN** and **Internet**.
data link level	Data link level is the communications protocol for the physical media-link used to transport the data.
datagram	A datagram is a message sent in a packet switched computer communications network. The message made up of source and destination addresses and the data itself. The datagram model implies that no connection, such as a virtual circuit is needed, to

send them and that they are not required to be delivered reliably or in sequence. See also **packet**.

DDN — Defense Data Network. The Defense Data Network (DDN) is a set of communications capabilities which links together computer systems within the Department of Defense (DoD). The DDN allows users of these computer systems to send mail and files between systems and to access other computers on the network in interactive terminal sessions. The DDN is part of the DARPA Internet. See **Internet**

daemon — A daemon is a UNIX system service. It is a program that is active in the background but not connected to a terminal.

destination — The destination address, an internet header field.

destination address — The destination address, usually the network and host identifiers.

flags — Flags is an internet header field carrying various control flags.

flow control — Flow control is the function and process of regulating the traffic and amount of data between flowing nodes so that neither node is sent more data than it can handle at a given time

fragment — A fragment is an IP packet that is one part of a UDP or TCP message. Messages may be split up, or fragmented, into fragments by IP if the message length exceeds the capability of the data link.

gateway — A software service installed at a switching node that connects two or more networks, especially if they use different protocols. A gateway provides UNIX internetworking with an extended logical network by transparently attaching one or more physical networks.

header — A header is the control information at the beginning of a message, segment, datagram, fragment, packet or block of data.

host — A host is a computer. In particular a source or destination of messages from the point of view of the communication network.

ICMP	Internet Control Message Protocol. ICMP is used by a gateway or destination host to communicate with a source host, for example, to report an error in datagram processing. ICMP, uses the basic support of IP as if ICMP were a higher level protocol, however, ICMP is actually an integral part of IP, and must be implemented by every IP module.
Identification	Identification is an Internet Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.
install	Install means to move the executable files from the distribution media to the system disk. In context, install can also mean to perform all the steps necessary to make a server or protocol operative
Internet	The Internet (spelled with initial capitalization) is the DARPA Internet System. See **DARPA**
Internet Address	Address is a source or destination address specific to the host level. It consists of a four octet (32 bit) source or destination address consisting of a Network field and a Local Address field.
Internet datagram	An internet datagram is the unit of data exchanged between a pair of internet modules (includes the internet header).
internet module	An internet module is an instance or individual implementation of the Internet Protocol, residing at a local host or gateway.
Internet Protocol	Internet Protocol (IP) is the network level protocol used by UNIX internetworking
internetwork	An internetwork is a supernetwork made up of two or more networks able to communicate with each other through gateways. See **gateway**
IP	See **Internet Protocol**
layer	A layer is a network function or set of related network functions that forms an autonomous functional block in the superset of network architectural functions. This method of partitioning the necessary network functions allows each layer to interface transparently to adjoining layers and thereby provides a method of making network

components more manageable.

Local Address — The Local Address the address of a host within a network. The actual mapping of an internet local address on to the host addresses in a network is quite general, allowing for many to one mappings.

local packet — A local packet is the unit of transmission within a local network.

machine — A machine is a host computer. The use of this term is similar to "host," and "node," but "machine" connotes the machine-specific or hardware aspects of the host computer, whereas "node" connotes the logical aspects of a network host. Host connotes the relationship of the local node machine to application systems and remote hosts.

module — A module is an implementation, usually in software, of a protocol or other procedure

network — A network is a collection of computer nodes able to communicate with each other

network interface — A network interface is the hardware and driver software that connects a host to a physical network.

octet — An octet is an eight bit byte.

OSI — (Open Systems Interconnection). OSI is a standard of the ISO. This standard attempts to provide for consistent hardware and software interfaces among network products. OSI and other standard setters such as IBM and the National Bureau of Standards generally divide network architecture into seven layers: physical, link, network, transport, session, presentation, and application.

packet — A packet is a package of data with a header which may or may not be logically complete. More often a physical packaging than a logical packaging of data.

point-to-point — Point-to-point is a network configuration in which two points are connected to each other by a dedicated line, which can be a direct cable connection, a leased line, or a dialup to a service providing dedicated lines

port — A port is the portion of a socket that specifies which logical input or output channel of a process is

associated with the data.

process — A process is a program in execution. A source or destination of data from the point of view of the TCP or other host-to-host protocol.

protocol — A protocol, in general, is a set of rules that enable a network entity to understand a communicating entity; however, the entity that employs these rules, such as the transport level protocol, TCP, is commonly referred to as a protocol. Therefore, a protocol is a software entity that implements a specific layer or function in a network architecture. In using the programmatic interface, a protocol is the next higher level protocol identifier, an internet header field.

RFC — Request For Comment. These refer to the "official" specification of Internet protocols and addressing mechanisms which are published by the Network Information Center of SRI in Palo Alto, CA and take the form of requests for comment.

root — Root is the login name of the super user. The super user is the user who has the widest form of machine privileges.

routing — Dynamic, or adaptive, routing is the ability to transfer data automatically to the destination node via alternative paths consisting of one or intermediate nodes. Routing includes the ability to ascertain available paths and to decide the best path, taking into account topology changes or node failures as they occur

server — A server is a system service, called a demon. It is usually a user program that runs in background, in user space, to provide a defined set of functions to the user who uses it through the command interface. Each time a user invokes it, the server provides a separate process for that user

socket — A socket is a file descriptor made up of system of data structures and pointers used by the kernel to identify and keep track of a process. It is an address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port. Sockets are transparent to the user.

Programs must open sockets to access network functions. Any one process cannot have more than 20 open files at a given time.

superuser — See **root**

TCP — Transmission Control Protocol is a transport level, connection-oriented protocol that provides reliable end-to-end message transmission over an internetwork

tuple — A tuple is a mathematical term for set of numbers composed of two or more factors. For example: [(XY)(AB)].

UDP — User datagram protocol is an unreliable user level transport protocol for transaction-oriented applications. It handles datagram sockets. It uses the IP for network services.

user — The user of the internet protocol. This may be a higher level protocol module, an application program, or a gateway program.

X.25 — X.25 is a circuit-switched network protocol used commonly in Europe and less so in the United States. X.25 is based on a three-layer, peer-communications protocol standard defined by the International Telegraph and Telephone Consultative Committee (CCITT).

Chapter 5

PROGRAMMER'S REFERENCE

NAME

intro – introduction to socket system calls and error numbers

SYNOPSIS

#include <sys/errno.h>

DESCRIPTION

This section describes all of the socket system calls used in System V STREAMS TCP. All of these system calls are accessible from the socket library, *libsocket*. The link editor *ld*(1) and the C compiler *cc*(1) search this library when the *-lsocket* option is specified. Most of these calls have one or more error returns, and some of the error codes are socket specfic. An error condition is indicated by an otherwise impossible return value. This is almost always −1; the individual descriptions specify the details. Note that a number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus *errno* should be tested only after an error has occurred.

The following is a complete list of the socket errors and their names as given in *<sys/errno.h>*. See the standard System V man page for *intro* for additional error codes.

63 ENOBUFS No buffer space available
: An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full. ENOBUFS is defined to have the same value as ENOSR.

90 TCPERR
: The starting value of socket related error codes.

TCPERR+0 EWOULDBLOCK Operation would block
: An operation that would cause a process to block was attempted on an object in non-blocking mode (see *fcntl*(2)).

TCPERR+1 EINPROGRESS Operation now in progress
: An operation that takes a long time to complete (such as a *connect*(2)) was attempted on a non-blocking object (see *fcntl*(2)).

TCPERR+2 EALREADY Operation already in progress
: An operation was attempted on a non-blocking object that already had an operation in progress.

TCPERR+3 ENOTSOCK Socket operation on non-socket
: Self-explanatory.

TCPERR+4 EDESTADDRREQ Destination address required
: A required address was omitted from an operation on a socket.

TCPERR+5 EMSGSIZE Message too long
: A message sent on a socket was larger than the internal message buffer or some other network limit.

TCPERR+6 EPROTOTYPE Protocol wrong type for socket
: A protocol was specified that does not support the semantics of the

socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

TCPERR+7 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system or no implementation for it exists.

TCPERR+8 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

TCPERR+9 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

TCPERR+10 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system or no implementation for it exists.

TCPERR+11 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use NS addresses with ARPA Internet protocols.

TCPERR+12 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

TCPERR+13 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

TCPERR+14 ENETDOWN Network is down
A socket operation encountered a dead network.

TCPERR+15 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

TCPERR+16 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

TCPERR+17 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

TCPERR+18 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

TCPERR+19 *unused*

TCPERR+20 EISCONN Socket is already connected
A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination when already connected.

TCPERR+21 ENOTCONN Socket is not connected
An request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket) no address was supplied.

TCPERR+22 ESHUTDOWN Can't send after socket shutdown

A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.

TCPERR+23 *unused*

TCPERR+24 ETIMEDOUT Connection timed out

A *connect* or *send* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

TCPERR+25 ECONNREFUSED Connection refused

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

TCPERR+26 EHOSTDOWN Host is down

A socket operation failed because the destination host was down.

TCPERR+27 EHOSTUNREACH Host is unreachable

A socket operation was attempted to an unreachable host.

TCPERR+28 ENOPROTOOPT Option not supported by protocol

A bad option or level was specified in a *getsockopt*(2) or *setsockopt*(2) call.

FILES

/usr/lib/libsocket.a

DEFINITIONS

Socket Descriptor

An integer assigned by the system when a socket is referenced by *dup*(2), or when a socket is created by *socket*(2), which uniquely identifies an access path to that socket from a given process or any of its children.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult *socket*(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

SEE ALSO

intro(3), perror(3).

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
accept	accept.2	accept a connection on a socket
adjtime	adjtime.2	correct system time to network average
bind	bind.2	bind a name to a socket
connect	connect.2	initiate a connection on a socket
getpeername	getpeernm.2	get name of connected peer
getsockname	getsocknm.2	get socket name
getsockopt	getsockopt.2	get and set options on sockets
listen	listen.2	listen for connections on a socket
recv	recv.2	receive a message from a socket
recvfrom	recv.2	receive a message from a socket
select	select.2	synchronous I/O multiplexing
send	send.2	send a message from a socket
sendto	send.2	send a message from a socket
setsockopt	getsockopt.2	get and set options on sockets
shutdown	shutdown.2	shut down part of a full-duplex connection
socket	socket.2	create an endpoint for communication

NAME

accept – accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(s, addr, addrlen)
int s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

Accept accepts a connection on a socket. The argument *s* is a socket which has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter which is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the "communications domain" [see *protocols*(4)]. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

RETURN VALUE

The call returns –1 on error. If it succeeds it returns a non-negative integer which is a descriptor for the accepted socket (*ns*, described above).

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.

SEE ALSO

bind(2), connect(2), intro(2), listen(2), socket(2), intro(7).

NAME

adjtime – correct the time to allow synchronization of the system clock

SYNOPSIS

```
#include <sys/time.h>

adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

DESCRIPTION

Adjtime makes adjustments to the system time, as returned by *gettimeofday*(3), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down. If *delta* is positive, the clock is set ahead. then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call. *N.B.* Implementations with poor control over the system clock will always return a zero timeval for olddelta.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The call *adjtime*(2) is restricted to the super-user.

RETURN VALUE

A return value of 0 indicates that the call succeeded. A return value of −1 indicates that an error occurred, and in this case an error code is stored in the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument points outside the process's allocated address space.

[EPERM] The process's effective user ID is not that of the super-user.

SEE ALSO

date(1), gettimeofday(3), rdate(1M), timed(1M), timedc(1M).
Timed Installation and Operation.
Berkeley Time Synchronization Protocol.

NAME

bind – bind a name to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with *socket*(2), it exists in a name space (address family) but has no name assigned. (Currently, only the Internet address family is supported.) *Bind* requests that *name* be assigned to the socket.

NOTES

The rules used in name binding vary between "communication domains" [see *protocols*(4)]. Consult the manual entries in Section 7 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of −1 indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

SEE ALSO

connect(2), getsockname(2), intro(2), listen(2), socket(2), inet(7), intro(7).

NAME

connect — initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

Connect initiates a connection on a socket. The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*; *namelen* is the length of *name*, which is an address in the address family of the socket. Each address family interprets the *name* parameter in its own way.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a −1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.

SEE ALSO

accept(2), getsockname(2), intro(2), socket(2), intro(7).

NAME

getpeername – get name of connected peer

SYNOPSIS

```
int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The interpretation of "name depends on the "communication domain" [see *protocols*(4)]. Only the Internet domain is currently supported.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), getsockname(2), intro(2), socket(2), intro(7).

NAME

getsockname — get socket name

SYNOPSIS

```
int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getsockname returns the current *name* for the specified socket *(s)*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return *namelen* contains the actual size of the name returned (in bytes).

RETURN VALUE

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), intro(2), socket(2), intro(7).

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

int setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in *<sys/socket.h>*, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle on/off local address reuse

SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output — not currently supported
SO_RCVBUF	set buffer size for input — not currently supported
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_PROTOTYPE	get/set the protocol number associated with the stream

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

With protocols that support out-of-band data, SO_OOBINLINE requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers

that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SO_PROTOTYPE binds a protocol number to the socket. This option is necessary to support raw IP sockets (for example) and, as such, is used primarily by the BSD compatibility module.

RETURN VALUE

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), socket(2), getprotoent(3).

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

listen — listen for connections on a socket

SYNOPSIS

int listen (s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a backlog for incoming connections is specified with *listen,* and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length to which the queue of pending connections may grow. If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED.

RETURN VALUE

A 0 return value indicates success; −1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(2), connect(2), socket(2).

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

recv, recvfrom − receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(s, buf, len, flags)
int s;
char *buf;
int len, flags;

int recvfrom(s, buf, len, flags, from, fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

DESCRIPTION

Recv and *recvfrom* are used to receive messages from a socket.

The *recv* call may be used only on a *connected* socket (see *connect*(2)), while *recvfrom* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see *socket*(2).

If no messages are available at the socket, the receive call waits for a message to arrive.

The *flags* argument to a send call is formed by *or*'ing one or more of the values:

```
#define MSG_OOB     0x1   /* process out-of-band data */
#define MSG_PEEK    0x2   /* peek at incoming message */
```

RETURN VALUE

These calls return the number of bytes received, or −1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

connect(2), intro(2), read(2), send(2), socket(2), intro(7).

NAME

select – synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

Select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set, i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

If a given file descriptor refers to a STREAMS device, a priority message on that stream is interpreted as an exceptional condition (the stream will also be ready for input). If a hangup or error has occurred on the stream, all conditions will be true.

RETURN VALUE

Select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EFAULT]	The memory pointed to by *readfds*, *writefds*, *exceptfds*, or *timeout* lies outside the valid address space for the process.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2), getdtablesize(3).

BUGS

Although the provision of *getdtablesize*(3) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of **<sys/socket.h>**.

Select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select* call.

Select will work on sockets, pseudo-terminals, generic tty devices, STREAMS devices, pipes, and regular files. It does not support arbitrary character or block devices.

NAME

send, sendto − send a message to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int send(s, msg, len, flags)
int s;
char *msg;
int len, flags;

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;
```

DESCRIPTION

Send and *sendto* are used to transmit a message to another socket (*s*). *Send* may be used only when the socket is in a *connected* state, while *sendto* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of −1 indicate some locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, then *send* blocks.

The *flags* parameter may be set to MSG_OOB to send "out-of-band" data on sockets which support this notion (e.g., SOCK_STREAM).

RETURN VALUE

The call returns the number of characters sent, or −1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

SEE ALSO

intro(2), recv(2), socket(2), intro(7).

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

int shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

SEE ALSO

connect(2), socket(2).

NAME

socket – create an endpoint for communication

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor.

s is a file descriptor returned by the *socket* system call.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The only currently supported format is AF_INET (ARPA Internet protocols).

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types include:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

Note that not all types are supported by all protocol families.

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_RAW sockets provide access to internal network protocols and interfaces. This type is available only to the super-user.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *protocols*(4).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with ETIMEDOUT as the specific code in

the global variable *errno*. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recv*(2), which returns the next datagram with its return address.

An *ioctl*(2) call can be used to specify a process group to receive a SIGUSR1 signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <*sys/socket.h*>. *setsockopt* and *getsockopt* [see *getsockopt*(2)] are used to set and get options, respectively. The following options are recognized at the socket level:

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle on/off local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output – not currently supported
SO_RCVBUF	set buffer size for input – not currently supported
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_PROTOTYPE	get/set the protocol number associated with the stream

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to

deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

With protocols that support out-of-band data, SO_OOBINLINE requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SO_PROTOTYPE binds a protocol number to the socket. This option is necessary to support raw IP sockets (for example) and, as such, is used primarily by the BSD compatibility module.

RETURN VALUE

A −1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT]
The protocol type or the specified protocol is not supported within this communication domain.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCESS] Permission to create a socket of the specified type and/or protocol is denied.

[ENOSR] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), intro(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), write(2), inet(7), intro(7).

NAME

intro – introduction to socket library functions

DESCRIPTION

This section describes functions that may be found in various libraries. The library functions are those other than the functions which directly invoke socket system primitives, described in section 2. All of these functions are accessible from the socket library, *libsocket*. The link editor *ld*(1) and the C compiler *cc*(1) search this library when the '–lsocket' option is given. The socket library also includes all the functions described in section 2.

FILES

/usr/lib/libsocket.a	the socket library

SEE ALSO

intro(2), cc(1), ld(1), nm(1).

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
bcmp	bstring.3	bit and byte string operations
bcopy	bstring.3	bit and byte string operations
bzero	bstring.3	bit and byte string operations
closelog	syslog.3	close system log file
endhostent	gethost.3	get network host entry
endnetent	getnetent.3	get network entry
endprotoent	getprotent.3	get protocol entry
endservent	getservent.3	get service entry
getdtablesize	getdtblsze.3	get descriptor table size
gethostbyaddr	gethost.3	get network host entry
gethostbyname	gethost.3	get network host entry
gethostname	gethostnam.3	get name of host
getnetbyaddr	getnetent.3	get network entry
getnetbyname	getnetent.3	get network entry
getnetent	getnetent.3	get network entry
getprotobyname	getprotoent.3	get protocol entry
getprotobynumber	getprotent.3	get protocol entry
getprotoent	getprotent.3	get protocol entry
getservbyname	getservent.3	get service entry
getservbyport	getservent.3	get service entry
getservent	getservent.3	get service entry
gettimeofday	gettimeday.3	get date and time
htonl	byteorder.3	convert values between host and network byte order
htons	byteorder.3	convert values between host and network byte order
index	string.3	string operations
inet_addr	inet.3	Internet address manipulation routines
inet_lnaof	inet.3	Internet address manipulation routines
inet_makeaddr	inet.3	Internet address manipulation routines
inet_netof	inet.3	Internet address manipulation routines
inet_network	inet.3	Internet address manipulation routines
insque	insque.3	insert/remove element from a queue
killpg	killpg.3	send signal to a process group
ntohl	byteorder.3	convert values between host and network byte order
ntohs	byteorder.3	convert values between host and network byte order

openlog	syslog.3	open system log file
perror	perror.3	system error messages
rcmd	rcmd.3	return a stream to a remote command
remque	insque.3	insert/remove element from a queue
rindex	string.3	string operations
rresvport	rcmd.3	return a stream to a remote command
ruserok	rcmd.3	return a stream to a remote command
setlogmask	syslog.3	control logging information
setnetent	getnetent.3	get network entry
setprotoent	getprotent.3	get protocol entry
setservent	getservent.3	get service entry
sys_errlist	perror.3	system error messages
sys_nerr	perror.3	system error messages
syslog	syslog.3	write mesage on system log file
wait3	wait3.3	non-blocking wait

NAME

bcopy, bcmp, bzero — bit and byte string operations

SYNOPSIS

```
bcopy(src, dst, length)
char *src, *dst;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

Bcopy copies *length* bytes from string *src* to the string *dst*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

BUGS

The *bcopy* routine takes parameters backwards from *strcpy*.

NAME

htonl, htons, ntohl, ntohs — convert values between host and network byte order

SYNOPSIS

```
#include <sys/types.h>
#include <sys/in.h>

netlong = htonl(hostlong);
unsigned long netlong, hostlong;

netshort = htons(hostshort);
ushort netshort, hostshort;

hostlong = ntohl(netlong);
unsigned long hostlong, netlong;

hostshort = ntohs(netshort);
ushort hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostent*(3) and *getservent*(3).

SEE ALSO

gethostent(3), getservent(3).

NAME

getdtablesize – get descriptor table size

SYNOPSIS

nfds = getdtablesize()
int nfds;

DESCRIPTION

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call *getdtablesize* returns the size of this table.

SEE ALSO

close(2), dup(2), open(2), select(2).

NAME

gethostbyname, gethostbyaddr, sethostent, endhostent, herror — get network host entry

SYNOPSIS

#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;
sethostent(stayopen)
int stayopen;

endhostent()

herror(string)
char *string;

DESCRIPTION

Gethostbyname and *gethostbyaddr* each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, *named*(1M), or broken-out fields from a line in **/etc/hosts**. If the local name server is not running these routines do a lookup in **/etc/hosts**.

```
struct  hostent {
        char    *h_name;          /* official name of host */
        char    **h_aliases;      /* alias list */
        int     h_addrtype;       /* host address type */
        int     h_length;         /* length of address */
        char    **h_addr_list;    /* list of addresses from name server */
};
#define h_addr  h_addr_list[0]    /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in h_addr_list; this is for backward compatibility.

When using the nameserver, *gethostbyname* will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable "HOSTALAIASES" contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See *hostname*(5) for the domain search procedure and the alias file format.

Sethostent may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*. Otherwise, queries are performed using UDP datagrams.

Endhostent closes the TCP connection.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine *herror* can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

h_errno can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error.
NO_DATA	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

FILES

/etc/hosts

SEE ALSO

resolver(3), hosts(4), hostname(5), named(1M).

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

```
int gethostname(name, namelen)
char *name;
int namelen;

int sethostname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is booted up.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* or *namelen* parameter gave an invalid address.

[EPERM] The caller tried to set the hostname and was not the super-user.

WARNING

Host names are limited to MAXHOSTNAMELEN (from <*sys/socket.h*>) characters, currently 64.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent — get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent ( )

struct netent *getnetbyname (name)
char *name;

struct netent *getnetbyaddr (net)
long net;

setnetent (stayopen)
int stayopen;

endnetent ( )
```

DESCRIPTION

getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, **/etc/networks**.

```
struct     netent {
     char  *n_name;          /* official name of net */
     char  **n_aliases;      /* alias list */
     int   n_addrtype;       /* net number type */
     long  n_net;            /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero-terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

getnetent reads the next line of the file, opening the file if necessary.

setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the network data base will not be closed after each call to *getnetent* (either directly, or indirectly through one of the other 'getnet' calls).

endnetent closes the file.

getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

SEE ALSO

networks(4).

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved. Only Internet network numbers are currently understood.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent — get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen;

endprotoent()

DESCRIPTION

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, **/etc/protocols**.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotobyname* or *getprotobynumber*.

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(4).

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getservent, getservbyport, getservbyname, setservent, endservent — get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservent ( )

struct servent *getservbyname (name, proto)
char *name, *proto;

struct servent *getservbyport (port, proto)
int port; char *proto;

setservent (stayopen)
int stayopen;

endservent ( )

DESCRIPTION

getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, **/etc/services**.

```
struct  servent {
    char *s_name;       /* official name of service */
    char **s_aliases;   /* alias list */
    int  s_port;        /* port service resides at */
    char *s_proto;      /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

getservent reads the next line of the file, opening the file if necessary.

setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the network data base will not be closed after each call to *getservent* (either directly, or indirectly through one of the other 'getserv' calls).

endservent closes the file.

getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3), services(4).

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gettimeofday – get date and time

SYNOPSIS

#include <sys/time.h>

```
gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday* call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks." If *tzp* is zero, the time zone information will not be returned.

The structures pointed to by *tp* and *tzp* are defined in *<sys/time.h>* as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest;     /* of Greenwich */
        int     tz_dsttime;         /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referenced invalid memory.

SEE ALSO

date(1), ctime(3)

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof — Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_addr(cp)
char *cp;

int inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard dot notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in dot notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the dot notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as **128.***net***.***host*.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A

network addresses as *net.host*.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "dot" notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostent(3), getnetent(3), hosts(4), networks(4), inet(7).

DIAGNOSTICS

The value −1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The string returned by *inet_ntoa* resides in a static memory area.

NAME

insque, remque — insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

Insque and *remque* manipulate queues built from doubly linked lists. Each element in the queue must in the form of "struct qelem". *Insque* inserts *elem* in a queue immediately after *pred*; *remque* removes an entry *elem* from a queue.

SEE ALSO

"VAX Architecture Handbook", pp. 228-235.

NAME

killpg — send signal to a process group

SYNOPSIS

killpg(pgrp, sig)
int pgrp, sig;

DESCRIPTION

Killpg sends the signal *sig* to the process group *pgrp*. See *signal*(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, or the sender must be the super-user.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Killpg will fail and no signal will be sent if any of the following occur:

[EINVAL]	*Sig* is not a valid signal number.
[ESRCH]	No process can be found in the process group specified by *pgrp*.
[ESRCH]	The process group was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and one or more of the target processes has an effective user ID different from that of the sending process.

SEE ALSO

kill(2).

NAME

perror, sys_errlist, sys_nerr — system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. The *perror* function of System V STREAMS TCP's socket library is similar to that in System V, but provides for the socket related error codes and associated messages in addition to the standard System V codes. When an application using *perror* is linked with the socket library (-lsocket), the System V STREAMS TCP *perror* is used by *ld*(1) instead of the standard *perror*.

First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno* (see *intro*(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

rcmd, rresvport, ruserok — routines for returning a stream to a remote command

SYNOPSIS

rcmd (ahost, inport, locuser, remuser, cmd, fd2p)
char **ahost;
unsigned short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

rresvport (port)
int *port;

ruserok (rhost, superuser, ruser, luser)
char *rhost;
int superuser;
char *ruser, *luser;

DESCRIPTION

Rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *Rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *Ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(1M) server (among others).

Rcmd looks up the host **ahost* using *gethostbyname*(3), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel and will also accept bytes on this channel as being signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostbyname*(3) routine, two user names and a flag indicating if the local user's name is the super-user. It then checks the files **/etc/hosts.equiv** and, possibly, **.rhosts** in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 0 is returned if the machine name is listed in the **hosts.equiv** file or if the host and remote user name are found in the **.rhosts** file; otherwise *ruserok* returns -1. If the *superuser* flag is 1, the checking of the **host.equiv** file is bypassed.

SEE ALSO

rcmd(1), rexecd(1M), rlogin(1), rlogind(1M), rshd(1M), rexec(3), hosts.equiv(4), rhosts(4).

BUGS

There is no way to specify options to the *socket* call which *rcmd* makes.

NAME

res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf,
buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, *exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets for use with Internet domain name servers. Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are stored a simple bit mask containing the bitwise "or" of the options enabled.

RES_INIT
: True if the initial name server address and default domain name are initialized (i.e., *res_init* has been called).

RES_DEBUG
: Print debugging messages. Only works if the resolver code has been built with the -DDEBUG option.

RES_AAONLY
: Accept authoritative answers only. With this option, *res_send* should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
: Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
: Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
: Set the recursion-desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, *res_mkquery* will append the default domain name to single-component names (those that do not contain a dot). This is the default.

RES_DNSRCH
: If this option is set, the standard host lookup routine *gethostbyname*(3) will search for host names in the current domain and in parent domains; see *hostname*(7).

Res_init reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried. *Res_mkquery* makes a standard query message and places it in *buf*. *Res_mkquery* will return the size of the query or −1 if the query is larger than *buflen*. *Op* is usually QUERY but can be any of the query types defined in *nameser.h*. *Dname* is the domain name. If *dname* consists of a single label and the RES_DEFNAMES flag is enabled (the default), the current domain name will be appended to *dname*. The current domain name is defined by the host-name or is specified in a system file; it can be overridden by the environment variable LOCALDOMAIN. *Newrr* is currently unused but is intended for making update messages.

Res_send sends a query to name servers and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned, or −1 if there were errors.

Dn_expand expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *Msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or -1 if there was an error.

Dn_comp compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. *length is the size of the comp_dn*. *Dnptrs* is a list of pointers to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is

compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

FILES

/etc/resolv.conf see resolver(4)

SEE ALSO

gethostent(3), named(1M), resolver(4), hostname(5).
RFC974, RFC1034, RFC1035.
Name Server Operations Guide for BIND .

NAME

rexec – return stream to a remote command

SYNOPSIS

rexec (ahost, inport, user, passwd, cmd, fd2p)
char **ahost;
unsigned short inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

rexec looks up the host **ahost* using *gethostbyname*(3), returning −1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a user name and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's **.netrc** file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call "getservbyname("exec", "tcp")" (see *getservent*(3)). The protocol for connection is described in *rexecd*(1M).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as *stdin* and *stdout*. If *fd2p* is non-zero, then a auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel and will also accept bytes on this channel as being signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rexecd(1M), rcmd(3), gethostent(3).

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NAME

strcasecmp, strncasecmp index, rindex — string operations

SYNOPSIS

```
strcasecmp(s1, s2)
char *s1, *s2;

strncasecmp(s1, s2, count)
char *s1, *s2;
int count;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings.

Strcasecmp and *strncasecmp* are identical in function to *strcmp(3)* and *strncmp(3)*, but are case insensitive. The returned lexicographic difference reflects a conversion to lower-case.

Index (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s* or zero if *c* does not occur in the string.

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

#include <syslog.h>

openlog(ident, logopt, facility)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

setlogmask(maskpri)

DESCRIPTION

Syslog arranges to write *message* onto the system log. The message is tagged with *priority*. The message looks like a *printf*(3) string except that **%m** is replaced by the current error message (collected from *errno*). A trailing new-line is added if needed. This message will be written to the system console, or the log file **/usr/adm/syslog** as appropriate.

Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, e.g., hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

If *syslog* cannot open the system log file, it will attempt to write the message on **/dev/console** if the LOG_CONS option is set (see below).

If special processing is needed, *openlog* can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *Logopt* is a bit field indicating logging options. Current values for *logopt* are:

LOG_PID	log the process id with each message: useful for identifying instantiations of daemons.
LOG_CONS	Force writing messages to the console if unable to append to **/usr/adm/syslog.** This option is safe to use in daemon processes that have no controlling terminal since *syslog* will fork before opening the console.
LOG_NDELAY	Open the connection to *syslogd* immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT Don't wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCLD, as *syslog* may otherwise block waiting for a child whose exit status has already been collected.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_USER Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_MAIL The mail system.

LOG_DAEMON System daemons, such as *ftpd*(1m), *routed*(1m), etc.

LOG_AUTH Messages involving user authorization.

LOG_LOCAL0 Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

Closelog can be used to close the log file.

Setlogmask sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

FILES

/usr/adm/syslog – the system log file

SEE ALSO

logger(1).

WARNING

/usr/adm/syslog must be created manually for logging to take effect. *Syslog(3)* will not create it.

NAME

wait3 — wait for process to terminate or stop

SYNOPSIS

```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid = wait3(status, options, rusage)
int pid;
union wait *status;
int options;
struct rusage *rusage;
```

DESCRIPTION

Wait3 is an alternate interface to *wait(2)* that allows non-blocking status collection. With a zero-valued *options* parameter, *wait3* functions similarly to *wait(2)*. Setting the *options* parameter to WNOHANG indicates that the call should not block if there are no processes that have status to report. A terminated child is discarded after it reports status. If *rusage* is non-zero, a summary of the resources used by the terminated process and all its children is returned. *N.B.* Currently, only the *ru_utime* and *ru_stime* fields of the *rusage* structure are filled in.

When the WNOHANG option is specified and no processes have status to report, *wait3* returns a *pid* of 0.

NOTES

A precise definition of the *status* parameter is given in *<sys/wait.h>*. See *signal*(2) for a list of termination statuses (signals); 0 status indicates normal termination. *status.w_stopval* is set to WSTOPPED (0177) if a process has not terminated and can be restarted; see *ptrace*(2). If the *w_coredump* field of the termination status is non-zero, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

RETURN VALUE

Wait3 returns −1 if there are no children not previously waited for; 0 is returned if WNOHANG is specified and there are no stopped or exited children. Otherwise, the process id of the reaped child is returned.

ERRORS

Wait3 will fail and return -1 immediately if the following is true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] WNOHANG was not set and *wait3* was interrupted due to receipt of a signal.

SEE ALSO

exit(2), times(2), wait(2).

NAME

intro – introduction to special files and protocols

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip_str.h>
#include <netinet/strioc.h>

DESCRIPTION

This section describes various special files and protocols that refer to specific System V STREAMS TCP/IP networking protocol drivers. Features common to a set of protocols are documented as a protocol family.

PROTOCOL FAMILY ENTRIES

A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per *socket*(2) type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in **socket**(2). A specific protocol may be accessed by creating a socket of the appropriate type and protocol family, by requesting the protocol explicitly when creating a socket, by executing the appropriate TLI primitives, or by opening the associated STREAMS device.

PROTOCOL ENTRIES

The system currently supports the DARPA Internet protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet and to the ICMP protocol. Consult the appropriate manual pages in this section for more information.

ROUTING IOCTLS

The network facilities provided limited packet routing. A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket-specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*:

```
struct rtentry {
        u_long  rt_hash;
        struct  sockaddr rt_dst;
        struct  sockaddr rt_gateway;
        short   rt_flags;
        short   rt_refcnt;
        u_long  rt_use;
        struct  ifnet *rt_ifp;
};
```

with *rt_flags* defined as follows:

```
#define RTF_UP          0x1     /* route usable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
#define RTF_DYNAMIC     0x10    /* created dynamically (by redirect) */
```

Routing table entries are of three general types: those for a specific host, those for all hosts on a specific network, and those for any destination not matched by entries of the first two types (a wildcard route). When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Normally the protocol specifies the route through each interface as a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (that is, the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOSR if insufficient resources were available to install a new route. User processes read the routing tables through the */dev/kmem* device. The *rt_use* field contains the number of packets sent along the route.

When routing a packet, the kernel will first attempt to find a route to the destination host. Failing that, a search is made for a route to the network of the destination. Finally, any route to a default ("wildcard") gateway is chosen. If multiple routes are present in the table, the first route found will be used. If no entry is found, the destination is declared to be unreachable.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

SOCKET IOCTLS

There are a few *ioctls* which have significance for the socket layer only. The *ioctl* call has the general form:

```
ioctl(so, code, arg)
```

SIOCPROTO

Enter a socket type into the kernel protocol switch table. The arguments used to create the socket used by this *ioctl* may be zero. The new socket type is downloaded by setting *arg* to a pointer to a specification block with the following structure:

```
struct socknewproto {
        int             family; /* address family (AF_INET, etc.) */
        int             type;   /* protocol type (SOCK_STREAM, etc.) */
        int             proto;  /* per family proto number */
        dev_t           dev;    /* major/minor to use (must be a clone) */
        int             flags;  /* protosw flags */
};
```

The flags currently supported are specified in the *<net/protosw.h>* header file as:

```
#define PR_ATOMIC        0x01    /* exchange atomic messages only */
#define PR_ADDR          0x02    /* addresses given with messages */
#define PR_CONNREQUIRED  0x04    /* connection required by protocol */
#define PR_RIGHTS        0x10    /* passes capabilities */
#define PR_BINDPROTO     0x20    /* pass protocol */
```

SIOCXPROTO
Purge the protocol switch table. The arguments used to create the socket used by this *ioctl* may be zero.

SIOCSPGRP
Set the process group for a socket to enable signaling (SIGUSR1) of that process group when out-of-band data arrives. The argument, *arg,* is a pointer to an **int** and, if positive, is treated as a process ID; otherwise, (if negative) is treated as a process group ID.

SIOCGPGRP
Get the process group ID associated with a particular socket. If the value returned to the **int** location pointed to by *arg* is negative, it should be interpreted as a process group ID; otherwise, it should be interpreted as a process ID.

SIOCCATMARK
Used to ascertain whether or not the socket read pointer is currently at the point (mark) in the data stream where out-of-band data was sent. If a 1 is returned to the *int* location pointed to by *arg,* the next read will return data after the mark. Otherwise (assuming out-of-band data has arrived), the next read will provide data sent by the client prior to transmission of the out-of-band signal.

FIONREAD
Returns (to the **int** location pointed to by *arg*) the number of bytes currently waiting to be read on the socket.

FIONBIO
Toggles the socket into blocking/non-blocking mode. If the **int** location pointed to by *arg* contains a non-zero value, subsequent socket operations that would cause the process to block waiting on a specific event will return abnormally with *errno* set to EWOULD-BLOCK; otherwise, the process will block.

QUEUE IOCTLS

Each STREAMS device has default queue high and low water marks, that can be changed by the super-user with the INITQPARMS specification in an **ioctl**(2). The ioctl is done on a driver or module, with the argument being an array of structures of type:

```
struct iocqp {
        ushort iqp_type;
        ushort iqp_value;
}
```

iqp_value specifies the value for the queue parameter according to **iqp_type,** which may be one of: **IQP_RQ**(read queue), **IQP_WQ**(write queue), **IQP_MUXRQ**(mux read queue), **IQP_MUXWQ**(mux write queue), or **IQP_HDRQ**(stream head queue), each OR'ed with either **IQP_LOWAT**(value is for low water mark of queue), or

IQP_HIWAT(value is for high water mark of queue).

INTERFACE IOCTLS

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, although certain interfaces such as the loopback interface, *lo*(7), do not.

The following *ioctl* calls may be used to manipulate network interfaces. The *ioctl* is made on a socket (typically of type SOCK_DGRAM) in the desired "communications domain" [see *protocols*(4)]. Unless specified otherwise, the request takes an *ifrequest* structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16];   /* name of interface (e.g. ec0) */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                int     ifru_metric;
                struct onepacket ifru_onepacket;
        } ifr_ifru;
#define ifr_addr        ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
#define ifr_metric      ifr_ifru.ifru_metric    /* routing metric */
#define ifr_onepacket ifr_ifru.ifru_onepacket /* one-packet mode params */
};
```

SIOCSIFADDR
Set interface address for protocol family. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
Get interface address for protocol family.

SIOCSIFDSTADDR
Set point to point address for protocol family and interface.

SIOCGIFDSTADDR
Get point to point address for protocol family and interface.

SIOCSIFBRDADDR
Set broadcast address for protocol family and interface.

SIOCGIFBRDADDR
Get broadcast address for protocol family and interface.

SIOCSIFFLAGS
Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

SIOCGIFFLAGS
Get interface flags.

SIOCSIFMETRIC
Set interface routing metric. The metric is used only by user-level routers.

SIOCGIFMETRIC
Get interface metric.

SIOCSIFONEP
Set one-packet mode parameters. The *ifr_onepacket* field of the *ifreq* structure is used for this request. This structure is defined as follows:

```
struct onepacket {
        int     spsize;         /* small packet size */
        int     spthresh;       /* small packet threshold */
};
```

One-packet mode is enabled by setting the IFF_ONEPACKET flag (see SIOCSIFFLAGS above). See *tcp*(7) for an explanation of one-packet mode.

SIOCGIFONEP
Get one-packet mode parameters.

SIOCGIFCONF
Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
    /* Structure used in SIOCGIFCONF request.
     * Used to retrieve interface configuration
     * for machine (useful for programs which
     * must know all networks accessible).
     */
struct  ifconf {
    int     ifc_len;                /* size of associated buffer */
    union {
            caddr_t ifcu_buf;
            struct  ifreq *ifcu_req;
    } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf   /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req   /* array of structures returned */
};
```

STREAMS IOCTL INTERFACE

Socket *ioctl* calls can also be issued using STREAMS file descriptors. The standard *strioctl* structure is used, with the *ic_cmd* field containing the socket *ioctl* code (from **<sys/socket.h>**) and the *ic_db* field pointing to the data structure appropriate for that *ioctl*, for all socket *ioctl*s except SIOCGIFCONF. For the SIOCGIFCONF *ioctl*, an *ifconf* structure is not used. Rather, the *ic_db* field points to the buffer to receive the *ifreq* structures.

TLI OPTIONS MANAGEMENT

Options may be set and retrieved in a manner similar to *getsockopt*(2) and *setsockopt*(2) using *t_optmgmt*(3N). Options are communicated using an options buffer, which contains a list of options. Each option consists of an option header and an option value. The *opthdr* structure gives the format of the option header:

```
struct opthdr {
        long level;     /* protocol level affected */
        long name;      /* option to modify */
        long len;       /* length of option value (in bytes) */
};
```

The option value must be a multiple of `sizeof(long)` bytes in length, and must immediately follow the option header. Following the option value

is the header of the next option, if present.

To get the values of options, set the *flags* field of the *t_optmgmt* structure to T_CHECK. It is not necessary to set the *len* fields in the option headers to the expected lengths of the option values, nor is it necessary to provide space between option headers for the option values to be stored (the *len* fields should be set to zero and the option headers should be adjacent). A new options buffer will be formatted and returned to the user. Note that T_CHECK may have failed even if *t_optmgmt* returns zero. The user must check the *flags* field of the returned *t_optmgmt* structure. If this field contains T_FAILURE, one or more of the options were invalid.

To set options, set the *flags* field of the *t_optmgmt* structure to T_NEGOTIATE.

To retrieve the default values of all options, set the *flags* field of the *t_optmgmt* structure to T_DEFAULT. For this operation, no input buffer should be specified.

NOTE

System V STREAMS TCP/IP man pages frequently cite appropriate RFCs (Requests for Comments). RFCs can be obtained from the DDN Network Information Center, SRI International, Menlo Park, CA 94025.

SEE ALSO

ioctl(2), socket(2), t_optmgmt(3N), tcp(7).

NAME

arp – Address Resolution Protocol

SYNOPSIS

3b2 master file configuration:

master.d/arp:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	arp	-	-	-	

master.d/arpproc:

m	-	app	-	-	-	

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers running the Internet protocols.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept. The ARP protocol is implemented by a STREAMS driver to do the protocol negotiation, and a separate STREAMS module to do the address translation.

To facilitate communications with systems which do not use ARP, *ioctls* are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each *ioctl* takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These *ioctls* may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure is as follows:

```
    /* ARP ioctl request */
struct arpreq {
    struct sockaddr         arp_pa;         /* protocol address */
    struct sockaddr         arp_ha;         /* hardware address */
    int                     arp_flags;      /* flags */
};
    /*  arp_flags field values */
#define ATF_COM          0x02    /* completed entry (arp_ha valid) */
#define ATF_PERM         0x04    /* permanent entry */
#define ATF_PUBL         0x08    /* publish (respond for other host) */
#define ATF_USETRAILERS  0x10    /* send trailer packets to host */
```

The address family for the *arp_pa sockaddr* must be AF_INET; for the *arp_ha sockaddr* it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM causes the entry to be permanent if the *ioctl* call succeeds. The peculiar nature of the

ARP tables may cause the *ioctl* to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The ATF_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

ARP watches passively for hosts impersonating the local host (that is, a host that responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

FILES

/dev/inet/arp

SEE ALSO

arp(1M), ifconfig(1M), inet(7).
RFC 826, RFC 893.

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

3b2 master file configuration:

FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	icmp	-	-	-	

programmer's interface:

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);
```

DESCRIPTION

ICMP is the error and control message (or device) protocol used by IP and the Internet protocol family. It may be accessed through a "raw socket" for network monitoring and diagnostic functions. The *proto* parameter to the socket call to create an ICMP socket is obtained from *getprotobyname* [See *getprotoent*(3).] ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;

[ENOSR] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL]
when an attempt is made to create a socket with a network address for which no network interface exists.

FILES

/dev/inet/icmp

SEE ALSO

send(2), recv(2), intro(7), inet(7), ip(7).

NAME

inet – Internet protocol family

SYNOPSIS

#include <sys/types.h>
#include <netinet/in.h>

DESCRIPTION

The Internet protocol family is a set of protocols using the *Internet Protocol* (IP) network layer and the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

ADDRESSING

Internet addresses are four-byte quantities, stored in network standard format. The include file *<sys/in.h>* defines this address as a discriminated union.

Sockets bound to the Internet protocol family use the following addressing structure:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

When using sockets, the *sin_family* field is specified in host order, and the *sin_port* and *sin_addr* fields are specified in network order.

Sockets may be created with the local address INADDR_ANY to affect wildcard matching on incoming messages. The address in a *connect*(2) or *sendto* [see *send*(2)] call may be given as INADDR_ANY to mean "this host." The distinguished address INADDR_BROADCAST is allowed as a shorthand for the broadcast address on the primary network if the first network configured supports broadcast.

When using the Transport Layer Interface (TLI), transport providers such as *tcp*(7) support addresses whose length varies from eight to sixteen bytes. The eight byte form is the same as a *sockaddr_in* without the *sin_zero* field. The sixteen byte form is identical to *sockaddr_in*. Additionally, when using TLI, the *sin_family* field is accepted in either host or network order. For communicating with other implementations via RFS, the preferred form is eight bytes with *sin_family* in network order.

PROTOCOLS

The Internet protocol family is comprised of the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction; UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The ICMP message protocol is accessible from a raw socket.

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use

the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the DARPA Internet may chose to use a single network number for the cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following *ioctl*(2) commands on a datagram socket in the Internet "communications domain"; they have the same form as the SIOCIFADDR command [see *intro*(7)].

SIOCSIFNETMASK
Set interface network mask. The network mask defines the network part of the address; if it contains more of the address than the address type would indicate, then subnets are in use.

SIOCGIFNETMASK
Get interface network mask.

SEE ALSO

rfsaddr(1M), ioctl(2), socket(2), intro(3), intro(7), icmp(7), ip(7), tcp(7), udp(7).

CAVEAT

The Internet protocol support is subject to change as the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported.

NAME

ip — Internet Protocol

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	ip	-	-	-	
fs	-	rip	-	-	-	

programmer's interface:

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

DESCRIPTION

IP is the network layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a "raw socket" or device when developing new protocols or special purpose applications.

A single generic option is supported at the IP level, IP_OPTIONS, that may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with *setsockopt* and examined with *getsockopt* [see *getsockopt*(2)]. The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets. *Proto* must be specified in *sockcf(4)*.

Outgoing packets automatically have an IP header prepended to them (based on the destination address given and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket has not been connected

[ENOSR] when the system runs out of memory for an internal data structure

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL] An unknown socket option name was given.

[EINVAL] The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

FILES

/dev/inet/ip /dev/inet/rip

SEE ALSO

getsockopt(2), send(2), recv(2), sockcf(4), intro(7), icmp(7), inet(7).

NAME

llcloop – software loopback network interface

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	loop	-	-	-	

programmer's interface:

```
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in sin;

s = socket(AF_INET, SOCK_XXX, 0);
.
.
.
sin.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (char *)&sin, sizeof(sin));
```

DESCRIPTION

The *llcloop* interface is a software loopback mechanism which may be used for performance analysis, software testing, and/or local communication. As with other network interfaces, the loopback interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl. The loopback interface should be the first interface configured, otherwise nameserver lookups for hostnames of other interfaces may fail.

FILES

/dev/llcloop

SEE ALSO

intro(7), inet(7).

NAME

slip – serial line IP network interface

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	sl	-	-	-	

DESCRIPTION

The *slip* interface is a driver that allows IP datagrams to be sent over normal serial lines. This is useful for connecting machines that do not have Ethernet hardware. As with other network interfaces, the slip interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl.

SEE ALSO

ifconfig(1M), slattach(1M), sldetach(1M), intro(7), inet(7).

NAME

socket – Socket Interface Driver

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
cs	-	sock	-	-	-	

DESCRIPTION

The *socket* driver is used to provide socket emulation to applications. Sockets are an alternate entry point into transport providers, such as *tcp*(7). The socket driver is a character device that acts as an alternate stream head, augmenting the functions of the standard stream head. It also provides support for miscelleanous functions such as *select*(2).

FILES

/dev/socksys

SEE ALSO

ifconfig(1M), intro(2), slattach(1M), sldetach(1M), intro(7), inet(7)

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	tcp	-	-	-	

programmer's interface:

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_STREAM, 0);

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses." Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using the *tcp* protocol are either "active" or "passive." Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, called "wildcard addressing," allows a single server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

TCP supports one socket option which is set with *setsockopt* and tested with *getsockopt* [see *getsockopt*(2)]. Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgment is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY (from *<netinet/tcp.h>*, to defeat this algorithm. The option level for the *setsockopt* call is the protocol number for TCP, available from *getprotobyname* [see *getprotoent*(3)].

Options at the IP transport level may be used with TCP; see *ip*(7). Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

TCP is also available as a TLI connection-oriented protocol via the special file **/dev/inet/tcp**. TCP options are supported via the TLI options mechanism.

TCP provides a facility, *one-packet mode*, that attempts to improve performance over Ethernet interfaces that cannot handle back-to-back packets. One-packet mode may be set by *ifconfig*(1M) for such an interface. On a connection that uses an interface for which one-packet mode has been set, TCP attempts to prevent the remote machine from sending back-to-back packets by setting the window size for the connection to the maximum segment size for the interface.

Certain TCP implementations have an internal limit on packet size that is less than or equal to half the advertised maximum segment size. When connected to such a machine, setting the window size to the maximum segment size would still allow the sender to send two packets at a time. To prevent this, a "small packet size" and a "small packet threshold" may be specified when setting one-packet mode. If, on a connection over an interface with one-packet mode enabled, TCP receives a number of consecutive packets of the small packet size equal to the small packet threshold, the window size is set to the small packet size.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one

[ENOSR] when the system runs out of memory for an internal data structure

[ETIMEDOUT] when a connection was dropped due to excessive retransmissions

[ECONNRESET] when the remote peer forces the connection to be closed

[ECONNREFUSED]
when the remote peer actively refuses connection establishment (usually because no process is listening to the port)

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated

[EADDRNOTAVAIL]
when an attempt is made to create a socket with a network address for which no network interface exists

FILES

/dev/inet/tcp

SEE ALSO

ifconfig(1M), getsockopt(2), socket(2), intro(7), inet(7), ip(7).

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

3b2 master file configuration:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
fs	-	udp	-	-	-	

programmer's interface:

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);

DESCRIPTION

UDP is a simple, unreliable datagram protocol that is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect*(2) call may also be used to fix the destination for future packets (in which case the *recv*(2), or *read*(2) and *send*(2), or *write*(2) system/library calls may be used). In addition, UDP is available as TLI connectionless transport via the special file **/dev/inet/udp**.

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (that is, a UDP port may not be "connected" to a TCP port). In addition, broadcast packets may be sent (assuming the underlying network supports this) by using a reserved broadcast address; this address is network interface dependent.

Options at the IP transport level may be used with UDP; see *ip*(7).

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected

[ENOSR] when the system runs out of memory for an internal data structure

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists

FILES

/dev/inet/udp

SEE ALSO

getsockopt(2), recv(2), send(2), socket(2), intro(7), inet(7), ip(7).

NAME

ttyp — pseudo terminal slave driver

vty — pseudo terminal master driver

SYNOPSIS

3b2 master file configuration:

master.d/ttyp:

*FLAG	#VEC	PREFIX	SOFT	#DEV	IPL	DEPENDENCIES/VARIABLES
ocst	-	ttyp	-	-	-	vty

master.d/vty:

ocs	-	vty	-	6	-	ttyp

programmer's interface:
TERMIO(7)

DESCRIPTION

The *ttyp* and *vty* drivers together provide support for a device-pair termed a *pseudo terminal*. A pseudo terminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes an interface identical to that described in *termio*(7). However, whereas all other devices which provide the interface described in *termio*(7) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

The following *ioctl* call applies only to pseudo terminals:

TIOCPKT
: Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD
: whenever the read queue for the terminal is flushed.

TIOCPKT_FLUSHWRITE
: whenever the write queue for the terminal is flushed.

TIOCPKT_STOP
: whenever output to the terminal is stopped a la ^S.

TIOCPKT_START
: whenever output to the terminal is restarted.

TIOCPKT_DOSTOP
: whenever *t_stopc* is ^S and *t_startc* is ^Q.

TIOCPKT_NOSTOP
whenever the start and stop characters are not ^S/^Q.

While this mode is in use, the presence of control status information to be read from the master side may be detected by a *select* for exceptional conditions.

This mode is used by *rlogin*(1) and *rlogind*(1M) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

FILES

/dev/ptyp[0-f][0-f]	master pseudo terminals
/dev/ttyp[0-f][0-f]	slave pseudo terminals

SEE ALSO

termio(7).

intel®

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

____
____
____
____

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

____
____
____
____

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

____
____
____
____

4. Did you have any difficulty understanding descriptions or wording? Where?

____
____
____
____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ______________

NAME ______________________________ DATE ______________

TITLE ______________________________

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________ PHONE (　　　) ______________

CITY ______________________ STATE ______________ ZIP CODE ______________
(COUNTRY)

Please check here if you require a written reply. ☐

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

INTEL CORPORATION
IMSO TECHNICAL PUBLICATIONS MS HF3-60
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9987

Please fold here and close the card with tape. Do not staple

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductor LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokoyo 15

NETHERLANDS
Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 SOLNA

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.M.B.H
Seidlestrasse 27
D-800 Munchen

UNIX®

SYSTEM V RELEASE 3.2

Intel UNIX® System V is the result of over seven years of operating system development with AT&T. The Intel UNIX software product family includes the base operating system, extensive development tools, networking capabilities and graphical user interfaces. Intel's commitment to quality products, support, and training make Intel UNIX System V the UNIX operating system of choice for today's application developers.

Intel's UNIX product line continues a long-standing tradition of products that adhere to established industry standards. Moreover, Intel UNIX products are tuned for Intel's 386™ and 486™ microprocessor architectures. Developers and users alike can feel confident that their investment in Intel UNIX products will give them an edge in today's competitive world.

Intel Corporation 465727-001

3065 Bowers Avenue, Santa Clara, California 95051